Automated Application of Fault Tolerance Mechanisms in a Component-Based System

Isabella Thomm Michael Stilkerich Rüdiger Kapitza Daniel Lohmann
Wolfgang Schröder-Preikschat
{ithomm,stilkerich,rrkapitz,lohmann,wosch}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany

ABSTRACT

Due to the reduction of structure sizes in modern embedded systems, tolerating soft errors presenting itself as bit flips becomes a mandatory task even for moderate critical applications. Accordingly, software-based fault tolerance mechanisms recently gained in popularity and a multitude of approaches that differ in the number and frequency of tolerated errors as well as their associated overhead have been proposed. As a consequence, an application- and environment-tailored selection of mechanisms is required to balance protection and costs.

Accounting the diverse solution space, we propose to make software-based fault tolerance a matter of configuration that should be transparent to the applications. While this would be cumbersome when using an unsafe programming language, we show that in the context of KESO, a JVM for deeply embedded systems, this can be achieved by utilizing the Java type system and static code analysis. As an initial technique we decided to add redundant execution to KESO, which enables us to selectively and transparently replicate an application. This essentially builds a first step to a JVM, which offers reliable execution of components as demanded by the system configuration. [1]

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Reliability, Design, Languages

[1]This work was partly supported by the German Research Foundation (DFG) under grants no. KA 3171/2-1 and LO 1719/1-1

JTRES 2011 September 26-28, 2011, York, UK

Keywords

KESO, Java, embedded systems

1. INTRODUCTION

The ability to tolerate soft errors has become an important aspect in safety-critical embedded systems. Such errors occur randomly and have – in contrast to hard errors – only a temporary effect on logical circuits or memory. Soft errors are a result of hardware failures that are becoming more likely to happen as a consequence of shrinking structure sizes. Also, they can be caused by extreme environmental conditions such as radiation [12] or current supply problems.

The developers of safety-critical systems are aware of these dependability issues as can be seen by the release of standards such as IEC 61508 or ISO 26262, which is widely spread in the automotive sector. These standards outline hardware-based redundancy and the employment of specialized error-correcting hardware components – such as ECC for memory devices or hardware watchdogs to recognize bogus behavior of components – as a possible solution.

However, this approach is often not feasible due to an immense cost pressure in this field, which manifests in the consolidation of the count of microcontrollers and the integration of mixed-criticality applications on a single electronic control unit (ECU). Besides the cost factor, hardware redundancy is also often impractical due to physical size, weight, and power constraints, which are eventually an essential requirement in unmanned flight, for example.

Software-based fault tolerance techniques such as replicated execution of code in space and time or monitoring software components pose a cheaper alternative to hardware-based techniques to increase system dependability and a combination of both approaches might also be sensible. Software-based approaches have already been employed in a wide range of safety-critical embedded systems, where the dominating programming language is C or Assembler. Due to the lack of type safety of C, application software has to be especially prepared for the use of many fault tolerance mechanisms – that are profoundly application-specific – such as control flow monitoring [4] or replicated execution. In case of replication, data distribution, voting and recovery mechanisms have to be manually inserted into the code base, since an identification of the data to be compared as well as the state of a replica is not possible in unsafe languages without additional support. By means of static analysis of the application – which is implemented in a type-safe language – and the support for software component isolation, fault tolerance mechanisms can automatically be applied as demanded by the safety

description without the need to change the application. This is particularly important, since reliability is a non-functional property that affects the entire software system and it is still an open question, which fault tolerance mechanisms are most suitable to mask certain hardware failures. Therefore, the kind of fault tolerance techniques and the location where to apply them should be a matter of configuration that is based on the safety requirements. These requirements are usually derived from test executions, where hardware faults to be tolerated are simulated. Also, the fault tolerance mechanisms themselves have to be tailored to the actual hardware platform and the application to be deployed by static code analysis to balance protection and costs.

We address these challenges by extending the KESO Java Virtual Machine (JVM) [27] – an ahead-of-time Java-to-C compiler for static software modules – to automatically weave fault tolerance measures into the C code as described in the system configuration. So far, KESO supports transparent N-redundant execution of application modules. This is a first step towards a JVM, that offers configurable fault tolerance for application software and is itself reliable.

The approach of providing redundant execution by KESO gains several benefits over a manual implementation in safety-critical control software in unsafe languages:

- Software-based *spatial isolation* in combination with hardware-based memory protection of replicas inhibits transient errors to affect code outside the replicated component.
- *Fault tolerance at option*: Configurable number of replicas, since this is depends on the number of transient faults that need to be tolerated, which is in turn related to the safety requirements of the application. Generally, the choice for a particular fault tolerance technique as well as how and the location where this technique is applied should be a configurable property. Initialization, activation (i.e., including data replication) and synchronization of replicas are performed by the KESO runtime system.
- *Automated fault tolerant replication*: Generation and invocation of application-specific voting functions as well as state recovery of replicas in case of the presence of transient errors due to type-safety of Java and ahead-of-time analysis of the application.
- *Enforcement of replica determinism*: Static analysis, to determine if code to be replicated satisfies certain criteria such as use of deterministic functions.

In the following sections, the system and fault model of a safety-critical control application, which shall be safeguarded by replicated execution, is described. Afterwards, we present the KESO JVM and how replicated execution can be integrated into this system including a discussion on possible solutions. In addition, the partially implemented approach is evaluated on a safety-critical control algorithm. This preliminary evaluation is followed by a discussion of related work in this field. To conclude this paper, we present our contributions and future work.

2. FAULT AND SYSTEM MODEL

Our fault model comprises transient hardware faults, that is, the focus is on bit flips in memory and logical circuits. The number of faults to be successfully recognized and corrected per processing interval is dependent on the respective fault detection and fault tolerance mechanisms. In case of software-based redundant execution, triple modular redundancy (TMR), for example, can tolerate a single affecting fault, that is, a 2n+1 replica system can tolerate n faults per processing interval. As single-bit flips account for over 90 percent of the soft error rate [11, 12], we concentrate on the occurrence of this fault type, though the extension to a multi-bit flip failure model is not restricted by our design. To assure a correct application of the redundant execution technique, several preconditions on the application and execution environment have to be satisfied, which are discussed in the remainder of this section.

2.1 Spatial and Temporal Isolation

The idea of the replication concept to gain fault tolerance is based on the supposition, that replicas are affected by faults independently and therefore requires replicated modules to be spatially isolated [19] by hardware components such as a memory protection unit. Replicas must not share common data and may only exchange information through safe communication channels. A faulty component must not corrupt the memory of other components and spread the error, which may result in failure of the entire system. It should be noted that purely software-based spatial isolation based on the type-safety of the programming language is not sufficient in this matter, since a bit flip may occur in a reference value and break the soundness of the type system.

Another essential requirement is temporal isolation, which is usually achieved by the employment of a real-time operating system that provides deadline monitoring. The OS monitors the execution time of a task and terminates the job signalling an error if the job exceeds its programmed worst-case execution time. A replica that is terminated in that way is considered as failed in the same way as a replica that returns a result different from the majority of replicas.

2.2 Application Model

The first assumption on the application components that we consider for replication is that the component has *run-to-completion* semantics, that is, it does not block during its cyclic execution interval. The second assumption in our current implementation is that replicated application components do not read from indeterministic sources (e.g., read input from hardware devices) or cause side effects outside the domain (e.g., by using certain operating system services). In case of replicated execution of the component to be safeguarded, the general activity flow of such a component starts at the acquisition of a well-defined data input (i.e., from sensors), which must be equal for each replica to ensure a deterministic computing. Thus, each replica maintains its own copied instance of that data. There are also redundant execution approaches, which are able to cope with slightly varying input data, but they are currently not in our focus. The replicated data is then used by all redundant modules and after the processing interval, the results of replicas are compared against each other by voters. Voters can be implemented in a variety of ways. Some implementations process primitive data types, where the values of the primitive input parameters are compared against each other, while others generate checksums to speed up the voting procedure. Trusting on result voting presumes that bit flips have an effect

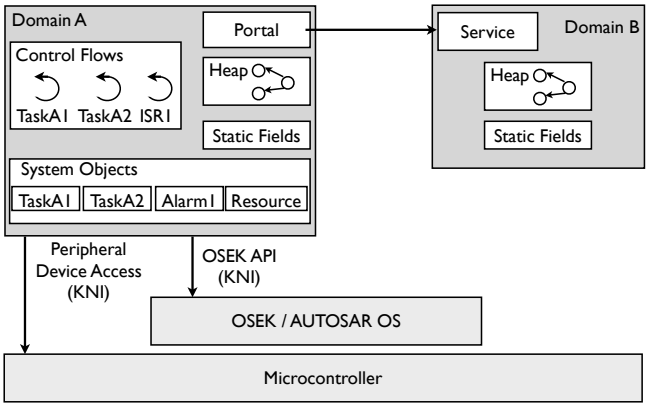


Figure 1: The KESO Architecture

on the result data. If, however, this precondition on the application does not apply, a replica can fail silently and affect the control procedure in a subsequent interval. For this scenario, voting procedures have to compare the entire state of the replicas. In case of a deviation in the result data or the replica states, the voters have to select a correctly processed replica, pick up its results, restore the failed replicas and propagate the correct values to the actuators. As can be seen, the manner in which the redundant execution mechanism shall be applied is highly application-specific.

In unsafe languages, programmers have to take care of the application of fault tolerance techniques on their software components by themselves. Therefore, we show in which way a static analysis on the application code and Java's type safety can be leveraged to identify the data to be compared as well as the entire state of the replicas.

3. DESIGN AND PROPOSAL

In this section, we introduce the KESO JVM, which is used as infrastructure software for the automated application of fault tolerance mechanisms. This is followed by a proposal, how replicated execution can be integrated into KESO as well as a presentation of possible solutions.

3.1 The KESO JVM

KESO [27] is an ahead-of-time Java compiler that generates ANSI C code from Java bytecode. The main goal of KESO is to provide software-based memory protection tailored towards the domain of embedded systems. KESO does not support all aspects of the Java language and the Java virtual machine and does not provide the full Java standard class library. In particular, KESO requires static applications and does neither support dynamic class loading nor Java reflection. The class library provided by KESO provides access to the system services of an OSEK/VDX [18] or AUTOSAR OS [2], which is currently presumed as system software; however, KESO is not limited to automotive applications.

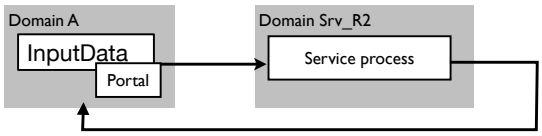
There is also a safe and lightweight mechanism to access device registers from Java code without affecting the type-safety of the program. KESO supports optional garbage collection for applications that want to use dynamic memory allocation.

KESO is a Multi-JVM, that is, it allows tasks to be spatially isolated in different protection domains, each of which appears as a JVM of its own from the application's point of view, as is depicted in Figure 1. A detailed description of KESO's memory protection mechanism and a discussion on how it compares to Java Isolates [8] can be found in a previous paper [27]. Isolation is constructively ensured by preventing any shared data among the different domains. This isolation is established based on the logical separation of the object heaps and by maintaining a separate set of the static fields in each domain. Each control flow (i.e., task or ISR) and all other system objects, such as events or locks, are statically assigned to a domain. A system object can only be accessed from other domains if explicitly permitted by the KESO system configuration. Java's type-safety guarantees, that an application can only access memory to that it has been given an explicit reference and the type of the reference also determines, how an application can access the memory area pointed to by the reference. To maintain the isolation, all inter-domain communication (IDC) mechanisms (i.e., portals and shared memory) must ensure that no reference values can be propagated to another domain. In addition to the software-based spatial isolation, KESO can actively support an AUTOSAR OS to provide hardware-based memory protection based on the use of a memory protection unit. KESO supports the OS by physically grouping the domain data (i.e., the *physically separated* heaps and static fields) in separate memory regions to recognize addressing errors and so to additionally harden the system [26].

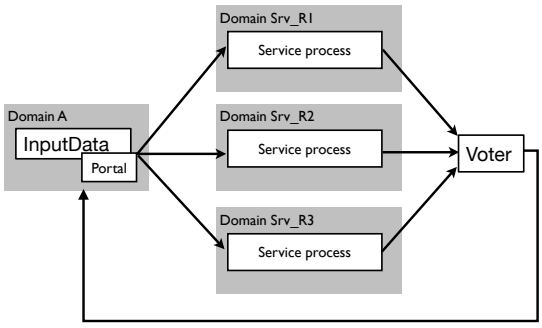
Portals, a synchronous remote method invocation mechanism, form the primary IDC mechanism in KESO. A domain can export a service with a system-wide unique name that can be imported and used by other domains. The relationships between client and service domains are statically specified in the configuration file. Only domains that explicitly import the service in the configuration file are able to use the service at runtime. A service export consists of a Java interface and an implementation of that interface. An instance of the implementation class, the *service object*, will be statically allocated by the compiler in the service domain. The compiler will create an anonymous proxy class that contains implementations of the interface's methods that perform a domain context switch and invoke the respective method on the service object in the domain that exports the service. A portal is an instance of this proxy class. The execution takes place in the context of the service domain, however, the control flow is the one that issued the portal call, which is migrated to the service domain for the duration of the portal call. Parameters and return values to a portal call are strictly passed by value to retain the heap separation property on which the spatial isolation is based. Primitive values are passed by value anyway. For reference parameters, a deep-copy of the referenced object graph is performed upon the portal call. The same happens when an object is returned as the result of a portal call. KESO provides a class-based mechanism to limit the amount of objects that are deep-copied. We elaborate on portals in detail in Section 4, as we have extended this IDC mechanism to support replicated services.

3.2 Replication

The first fault tolerance mechanism to be added to KESO is redundant execution, as this is a well-understood method to improve the dependability of a system. The sphere of replication in our approach is the domain, which provides



(a) Single Execution



(b) Replicated Execution

Figure 2: Replication Approach: The interface is identical in both cases.

many of the properties necessary for replication. We use KESO's portal mechanism as a transition point between the single and replicated execution. From the programmer's point of view, the replicated execution is transparently hidden beyond the interface of an exported service. In the following, we will step through the building blocks of our design.

3.2.1 Spatial Isolation

The occurrence of a soft error in a replica must not affect the execution of other replicas or software modules that run on the same microcontroller. Therefore, they must be isolated from each other. By replicating software components in a domain granularity, this spatial isolation is already provided. The runtime system inhibits a control flow inside a domain to leave the protection context or to reference memory locations outside the domain due to a bit flip that affects the application data only. Originally, the domain mechanism was intended to detect software bugs and a violation of the protection context led to a fatal exception, for example as a consequence of a failed null reference or array bounds checks.

For replicated execution, this approach has to be slightly modified to eventually induce a recovery of the domain state. In a mixed criticality system, KESO determines if an application is operating in normal or redundancy mode. In the latter mode, we have to check, if the same protection violation occurs in all replicated domains, which is a sign for a programming error in the software module. In such a case, we retain KESO's original behavior that leads to an abort of the respective application. Otherwise, the safety violation is caused by a soft error, which must result in a recovery of the faulty domain. However, reference values can also be affected by a fault. For this case, we apply hardware-based memory protection to the replicas in KESO domains to contain such a fault in the affected domain.

3.2.2 Fault Tolerance at Option

A software component to be safeguarded is replicated via the KESO configuration file by multiple instantiation of that component in own domains. Resulting from the Multi-JVM architecture, it is easily possible to create multiple instances of a domain, since all the state belonging to a domain will be re-instantiated. Creating multiple instances of a domain is only a matter of adapting the system configuration file. The KESO runtime system calls the constructors of the defined replica classes and initializes the respective heaps and static fields in each domain separately. In Figure 2, we have exemplified the setup of a TMR system.

3.2.3 Automated Fault Tolerant Replication

The portal mechanism is deployed for the data transfer between domains, that is, the domains Srv_R1, Srv_R2 and Srv_R3 are configured to export the processing function to be replicated as a named service, while the activating domain A imports this service. Since service names are unique in a KESO system, KESO's compiler can recognize replicated services by definitions of a service of the same name within multiple domains. In order to support service replication, the portal mechanism has to be modified to trigger an invocation of the service in all replicas. We will elaborate on that in more detail in Section 4.

The portal parameters represent the input data to the replicas, while the return value delivers the correct output data, which are further processed by domain A after having been processed by a majority voter. Due to the *by-value* semantics on portal parameters, each domain is provided with an own copy of the input data.

As soon as all replicas have finished, a voting function has to decide over their correct execution. Several alternatives for such a function exist and can easily be integrated into KESO. Generally, there are two approaches, how voting can be performed:

- Voting over the output data of replicas is sufficient for many control applications. The precondition on this variant is that a bit flip in a replica has to directly affect the results of the computation, that is it needs to become visible in the return value of the service method or has to trigger a memory or timing protection violation that is trapped by the OS. The advantage of that variant is the fast operation of the voters. A drawback is that replicas can fail silently, if the bit flip does not corrupt the output data and can have an effect in a later interval. This can be true for domains using a GC, where a bit flip occurs during the collection phase. A GC, which can cope with transient errors, might be used for this approach.

- Voting over the entire domain state. This technique seems inappropriate at a first glance, however, embedded control applications usually have a relatively small state, outlining this approach as another possible solution, which we will evaluate in the future.

A static analysis of the application can also help to identify which voting variant is more sensible to the respective application.

According to the selected approach, the appropriate voting functions are generated by the compiler, fitting the return type of the service and the number of replicas. In case of a voting error, the faulty component is reset (i.e., clearing the heap) and reinitialized by transferring the complete state of a healthy domain (copying the objects reachable in the healthy domain). For this, we leverage Java's type-safety

and copy all static and non-static class fields as well as the heap. The amount of time, which is necessary to recover a domain has to be considered in the system schedule, so that no replica misses its deadline. Some transient errors – such as a manipulation of the break condition in a loop – may also cause the application to exceed its time slice. Usually, the incorporated RTOS terminates that application. If the time for domain recovery is considered in the schedule, the RTOS can trigger the KESO runtime system to induce the state recovery.

3.2.4 Enforcement of Replica Determinism

Safety-critical control components often exhibit the *run-to-completion* model to eliminate any source of unpredictable behavior as discussed in Section 2. In KESO, we can enforce replica determinism by static analysis of the application code to identify certain points of indeterminism, which are manifold. Replicas are synchronously activated by portal calls and are synchronously terminated by voting to deliver a correct output when the portal returns. While using a service in non-replicated mode in more than one domain is not an issue, care must be taken in case of redundant execution. A precondition on code replication techniques is that replica invocations must be ordered if invoked from multiple control flows. Our current prototype evades this issue by limiting calls to replicated services to a single client control flow only. Allowing for multiple activation sources may cause indeterministic behavior of the replica. We leave an investigation of this issue and possible solutions as future work.

Another major source of non-determinism poses communication with other components outside the sphere of replication. The issues of communication are two-fold. On the one hand, a transient error must be contained in the replicated domain. A critical use of the portal service in case of replicated mode is, for example, the transfer of a value into another domain, where the application in that domain depends on that value. If a bit flip happens to the portal parameter, the error is spread beyond domain boundaries. On the other hand, any data from outside the sphere of replication can cause diversity in the redundant execution. These code locations within the replica can be identified due to well-known interfaces and comprise

- native interface function calls, use of system services and peripheral device access through KESO's raw memory abstractions
- shared memory usage, which must be manually synchronized using resources (i.e., by means of the immediate priority ceiling protocol)
- Inter-domain communication with replicated as well as non-replicated domains via portals

At the current state, KESO's compiler will signal an error if the reachability analysis detects use of any of the above mechanisms that are known to cause side-effect by a replicated service. This supports replication of applications as suggested by [19].

We plan, however, to extend this mechanism for replicated applications, which explicitly want to communicate with other components. With respect to the schedule, the replication threads must be merged at the identified synchronization points to assure a correct execution of replicas by

- Voting: Data that is transferred to other components must be compared, so that faults are contained in the domain.
- Replication: Data input (further read of sensor values or return values of native calls, for example) must be copied, so that it is equal for all replicas.

In related fault tolerance projects, using indeterministic functions is a known issue. The Replicant [20] system, for instance, introduced *relaxed determinism*, that is, it loosely replicates the order of events in replicas. The system uses annotations from the application developer to mark certain points of indeterminism and *synchronize syscall* hints to suppress this relaxed determinism, if necessary.

Another example is the AUTOSAR platform, which incorporates the Virtual Function Bus (VFB) [3] concept. It facilitates communication of applications independent from the underlying ECU and network setup. The VFB does neither support fault tolerant redundant execution nor spatial isolation. However, due to the VFB specification, an implementation of the VFB – the so-called Runtime Environment (RTE) – has to provide *implicit* read and write operations, so that runnable entities (i.e., schedulable functions within applications) do not have to directly access message buffers. Some legacy control algorithms are not able to cope with multiple explicit access of those buffers and it might corrupt the computation. The solution for this issue is related to the indeterministic read of sensor values. Basically, implicit VFB functions copy a specific value from a message buffer, which is valid for one processing interval of the runnable. Any implicit read or write to a data item is applied to a copy, which is written once the interval has finished. However, the VFB specification does neither tell when nor how to apply implicit operations.

In KESO, each replica holds a proxy object for the data input at synchronization points, which are identified during static analysis. A single replica then invokes the native call. Such a replica is selected due to a voting over the current domain state data. A data flow examination might eventually help to restrict the set of data to be compared in order to speed up synchronization point handling. The result of the native call invoked by the chosen replica is then copied to all proxy objects so that the redundant execution of all replicas can continue.

In general, start and end locations of replicated services as well as synchronization points in services that communicate with other components outside their domain pose single points of failure (SPOF). A technique that can handle those SPOFs by means of coded computing has been developed at our research group and is currently under review.

4. IMPLEMENTATION

Control software can be split into parts that are controlled by replication and those which are not. This is the same approach as preparing software for software-based memory protection without any replication. The parts of the application to be executed in non-replicated mode as well as those parts to be safeguarded are configured to reside in different domains in the KESO system configuration. The domains Srv_R1, Srv_R2 and Srv_R3 export the service PROCESS, which shall be processed redundantly, whereas the activating client application in domain A statically imports this service.

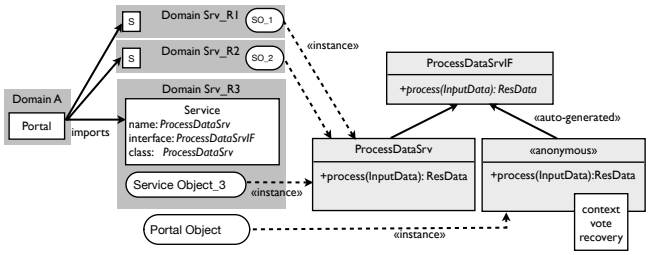


Figure 3: Service replication via portals

```
public void bar(InputData sensorData) {
  // service retrieval
  ProcessDataSrvIF srv = (ProcessDataSrvIF)
    PortalService.lookup("ProcessDataSrv");
  // replicated execution
  ResData result = srv.process(sensorData);
  // replicated execution finished
  result.actuate();
}
```

Figure 4: Using the portal service for replication

The KESO runtime system automatically creates multiple instances of the service object, one in each of the replica domains. Each replicated domain is initialized by KESO and has its own static fields and heap.

For service replication, we leverage the portal mechanism of KESO, which is depicted in Figure 3. A service is defined by a unique name, which is used by the client domain to refer to that service, a Java interface, and the implementation of that interface. If the software already uses software-based isolation to protect the important part from other less critical code sections, the application does not need to be touched at all. In the other case, the regular method call that usually starts the control procedure has to be replaced by a portal call. If neither replication nor isolation is required, the portal call is optimized to a regular call by *jino*, thus erasing any overhead.

4.1 Service Replication

A service is replicated by means of a portal call as shown in Figure 4. The starting point is the auto-generated anonymous class, that is used by the client domain through the portal object, which is returned by the `PortalService`'s `lookup()` function. The generated class implements the service interface and stub functions that perform the protection context switch. Replicated and non-replicated services need to be distinguished from each other, which can be determined by the number of domains, the same service (identified by its unique name) is defined in. For replicated services, we extend the generation of the stub functions to invoke the service of all replicas and to vote on the returned values. The result of the voting process is then returned as the result of the portal call.

4.2 Service Synchronization

For a fault tolerant prototype, we have chosen a simple voting variant, which votes only on the returned result of a service method in the replica domains. The advantages and disadvantages of that approach were discussed in Section 3.2.1 and we consider them as sufficient for our evaluation scenario. The return value of the portal services denotes the output value, so *jino* analyzes the return types and creates the appropriate voting functions. The voters operate on primitive data types, where the values of the primitive input parameters are compared against each other and the voter function signals whether there is a conflict or not. In case of a primitive return value of the portal call, the determination of the voter type is straight-forward. For complex data types, the building tool retrieves the necessary information by traversing the class fields of the return type, which can either be primitive, an array or another complex data type. To avoid cyclic dependencies, we have to memorize which objects have already been analyzed. The process is similar to the scanning phase of the garbage collector. In this way, all needed types of voters can be determined and invoked. An execution of the voter function picks a successfully processed replica and returns the output value to the portal call. If the voter detects a difference in a replica in contrast to the other ones, it triggers the recovery for the faulty replica and selects an output value from the succeeded remaining replicas to be returned to the caller.

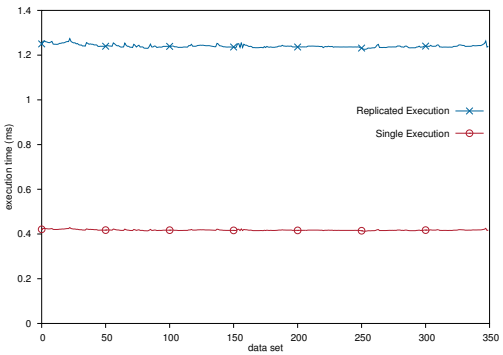
5. PRELIMINARY EVALUATION

In our first experiments, we evaluate the overhead added by our replication mechanism. As an example application, we use the flight-attitude controller component of the I4Copter [28] framework, a control application for a quadrotor helicopter. The flight-attitude controller component is responsible for keeping the aircraft at a certain angle with respect to a reference point by appropriately adjusting the thrust levels of the four rotors. The controller is input with values from various sensors such as gyroscopes and accelerometers. The output of the controller is four engine thrust levels. This component is part of a real-world safety-critical system.

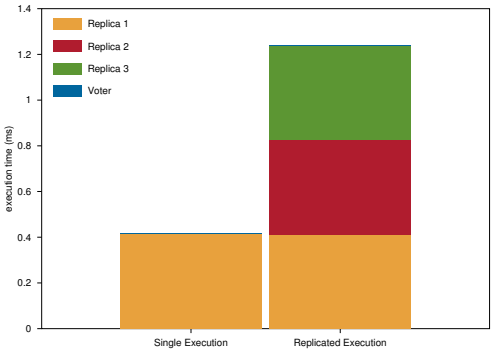
We run our experiments on an Infineon TriCore TC1796 microcontroller clocked at 150 MHz. Our board is equipped with 2 MiB of Flash ROM and 1 MiB of SRAM. We use CiAO [9] as OS, an own AUTOSAR OS implementation. CiAO supports hardware-based memory protection using a region-oriented memory protection unit and deadline monitoring for tasks. To measure the runtimes, we read the value of the TC1796's free-running 75 MHz system timer before and after the measured code section.

The component exhibits the properties required by our current prototype: It has run-to-completion semantics and performs internal computation only, that is the execution of the component has no side effects outside the domain. In a flying configuration, the controller is periodically executed each 9 ms. The component exports a service that is provided with the sensor values of the current period and returns an object containing the thrust-levels ready to be actuated. To provide reproducible behavior, we feed the controller with compiled-in sensor data taken from a flight data log.

Figure 5(a) shows the execution time for a single execution and a three-times replicated execution of the controller for 350 iterations. As depicted the controller has an almost constant execution time. The single execution variant uses an unreplicated portal service for the controller, which imposes that the input data are copied when the portal is invoked. The replicated variant comprises three instances of the controller component. The generated proxy method of the portal



(a) Execution Times



(b) Execution Time Breakdown

Figure 5: Replicated Flight-Attitude Controller

object creates three copies of the input data, one in each replica, and sequentially invokes the portal service in each replica using the corresponding set of input data. Finally a voting on the returned result is performed. As copying the input data is also part of an unreplicated portal service execution, the overhead added by the replicated execution is the cost of voting.

Figure 5(a) shows that the replicated variant requires 3x the time of the unreplicated variant, which is the expected result. We investigated the execution times for each of the replicas separately and additionally the cost of the voting process. The result is shown in Figure 5(b). The three replicas have a runtime identical within the accuracy of the measurement of 413 μs. The single execution has a slightly higher runtime of 416 μs. The voting process itself takes 1.6 μs.

The execution time overhead for the initial creation of the replicas is negligible. The replica are statically created by the compiler and initialized in the startup phase. The initialization consists only of few operations, most notably the initialization of the heap management data structures.

6. RELATED WORK

Traditionally, handling of soft errors is addressed by safety critical applications that are executed in harsh environments such as found in the aerospace domain. Here, the high costs of hardware-centric approaches like double or triple hardware redundancy [10, 25, 30, 31] can be tolerated.

To reduce costs and make soft error tolerance available to a wider range of application scenarios, software-based fault handling methods for control flow [15, 16, 14, 17, 29, 22, 6] or memory protection [23] were developed over the recent years. Lately, Chang et al. [21] proposed SWIFT-R, an approach that triplicates machine instructions and uses a majority voter in combination with encoding techniques.

Unlike these approaches that operate at the level of machine instructions we propose tolerating soft errors at the level of components. This avoids dependence on a custom compiler for the target platform and enables to selectively protect critical parts of applications.

There are several approaches that apply replication at a higher level of abstraction. In case of Shye et al. an approach is presented that replicates processes in a standard operating system [24]. While transparent application of software-fault tolerance is possible this way it is rather coarse grained and only a single technique is used.

More in line with our vision of a soft error hardening of applications is the work of Afonso et al. [1], that selectively enhances an embedded real-time system with fault tolerance mechanisms by using aspect-oriented programming (AOP). In contrary to Afonso who addresses a typical C++ application on an embedded system and therefore requires extensive knowledge about the application when applying fault tolerance mechanisms our approach strongly benefits from the modular design of KESO applications.

At the level of Java virtual machines the work of Napper et. al [13] and Friedman et. al [7] focus on the replication of a virtual machine as a whole. Their work is dedicated towards tolerating fail stop faults in a distributed setting but does not address soft errors on a single system.

7. CONCLUSION

In this paper, we presented how the N-redundant execution technique can be integrated into a Java ahead-of-time compiler. Currently, a full support of the proposed replicated execution and variants – such as the *pair-and-spare* technique – is in progress. Fault tolerance methods impose more overhead in space and time to improve the system reliability by their nature. However, a KESO application – which can also be a device driver, for example – can select these methods according to the safety requirements and only pays for the fault tolerance features that it actually needs. Functional safety is a non-functional property that should be transparent to the applications and the required level of safety should be a matter of configuration. *Static analysis* on software components that are implemented in any *type-safe* language and the support for *spatial isolation* facilitate to automatically apply configurable fault tolerance techniques that are tailored to the application's requirements. As future multicore embedded systems will become less reliable due to shrinking structure sizes, the available parallelism of these platforms can also be used to compensate the overhead imposed by the application of fault tolerance techniques.

8. FUTURE WORK

Our next steps will be to integrate more fault tolerance measures into KESO to be applied automatically as demanded by the safety configuration and to evaluate them with respect to performance and footprint on the characteristics of static embedded applications. We also have to test the effectiveness of our approach with respect to tolerated and undetected errors by means of fault injection. This comprises

an evaluation on several voting techniques including the consideration of application-specific SPOFs, automated control flow monitoring – as control flow errors make up 33%-77% of all runtime errors depending on the application [5] – and the use of checksums. We will also analyze the use of a more fine-grained level of redundancy, that is, a software component does not have to be replicated as a whole, but only certain parts of it. This helps to determine the best possible rate between memory and execution time costs with its effects on dependability in mind. Transient errors can affect user applications as well as system services, that is, also the OS or the KESO runtime system can be compromised by bit flips. For this, fault tolerance techniques have also to be applied to the virtual machine itself, which comprises the use of a fault tolerant GC, for example. As we want to rely on existing software standards and since there is currently no implementation of a type-safe AUTOSAR OS, which can benefit from KESO directly, the OS itself has to take care of soft errors.

9. REFERENCES

[1] F. Afonso, C. Silva, N. Brito, S. Montenegro, and A. Tavares. Aspect-oriented fault tolerance for real-time embedded systems. In *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software (ACP4IS '08)*, pages 1–8, 2008.

[2] AUTOSAR. Specification of operating system (version 3.0.2). Technical report, Automotive Open System Architecture GbR, June 2008.

[3] AUTOSAR. Specification of the virtual functional bus (version 3.0.7). Technical report, Automotive Open System Architecture GbR, July 2010.

[4] AUTOSAR. Specification of watchdog manager (version 2.1.0). Technical report, Automotive Open System Architecture GbR, Oct. 2010.

[5] S. Chen, X. Hu, B. Liu, and J. Chen. An on-line control flow checking method for vliw processor. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pages 248–255, Washington, DC, USA, 2007. IEEE Computer Society.

[6] C. Fetzer, U. Schiffel, and M. Suesskraut. AN-Encoding compiler: Building safety-critical systems with commodity hardware. In *Computer Safety, Reliability, and Security*, volume 5775 of *Lecture Notes in Computer Science*, pages 283–296. Springer Berlin / Heidelberg, 2009.

[7] R. Friedman and A. Kama. Transparent fault-tolerant java virtual machine, 2003.

[8] JSR 121: Application Isolation API Specification. Sun Microsystems JCP, June 2006.

[9] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX ATC*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX.

[10] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37:160–174, February 1988.

[11] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong. Characterization of multi-bit soft error events in advanced srams. In *Electron Devices Meeting, 2003. IEDM '03 Technical Digest. IEEE International*, pages 21.4.1 – 21.4.4, Dec. 2003.

[12] D. Makowski. *The Impact of Radiation on Electronic Devices with the Special Consideration of Neutron and Gamma Radiation Monitoring*. Dissertation, Technical University of Lodz, 2006.

[13] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant java virtual machine. In *In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium*, pages 425–434, 2002.

[14] N. Oh, S. Mitra, and E. McCluskey. Ed4i: Error detection by diverse data and duplicated instructions. *Computers, IEEE Transactions on*, 51(2):180–199, 2002.

[15] N. Oh, N. Shirvani, and E. P.P. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51:63–75, 2002.

[16] N. Oh, P. Shirvani, and E. McCluskey. Control-flow checking by software signatures. *Reliability, IEEE Transactions on*, 51(1):111 –122, Mar. 2002.

[17] J. Ohlsson and M. Rimen. Implicit signature checking. *Fault-Tolerant Computing, International Symposium on*, 0:0218, 1995.

[18] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, Feb. 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2011-08-17.

[19] S. Poledna. Replica determinism in distributed real-time systems: a brief survey. *Real-Time Systems Journal*, 6(3):289–316, 1994.

[20] J. Pool, I. Sin, K. Wong, and D. Lie. Relaxed determinism: Making redundant execution on multiprocessors practical. In *In Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS)*, 2007.

[21] G. A. Reis, J. Chang, and D. I. August. Automatic instruction-level software-only recovery. *IEEE Micro*, 27(1):36–47, 2007.

[22] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *In Proceedings of the 3rd International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[23] P. P. Shirvani, N. Saxena, S. M. Ieee, E. J. Mccluskey, and L. F. Ieee. Software-implemented edac protection against seus. *Reliability, IEEE Transactions on*, 49:273–284, 2000.

[24] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *IEEE Trans. Dependable Secur. Comput.*, 6:135–148, April 2009.

[25] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. Ibm's s/390 g5 microprocessor design. *IEEE Micro*, 19:12–23, March 1999.

[26] M. Stilkerich, J. Schedel, P. Ulbrich, W. Schröder-Preikschat, and D. Lohmann. Escaping the bonds of the legacy: Step-wise migration to a type-safe language in safety-critical embedded systems.

In G. Karsai, A. Polze, D.-H. Kim, and W. Steiner, editors, *14th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '11)*, pages 163–170, Washington, DC, USA, Mar. 2011. IEEE.

[27] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat. Tailor-made JVMs for statically configured embedded systems. *Concurrency and Computation: Practice and Experience*, 2011. To appear. http://dx.doi.org/10.1002/cpe.1755.

[28] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11)*, pages 380–396, New York, NY, USA, 2011. ACM.

[29] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray. Low-cost on-line fault detection using control flow assertions. *IEEE International On-Line Testing Symposium*, 0:137, 2003.

[30] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the IEEE Aerospace Applications Conference*, volume 1, pages 293 –307 vol.1. IEEE, Feb. 1996.

[31] Y. C. B. Yeh. Design considerations in boeing 777 fly-by-wire computers. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE '98)*, 1998.